Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/6180--01-8588

Supervisory Control System for Ship Damage Control: Volume 7 — Software Architecture

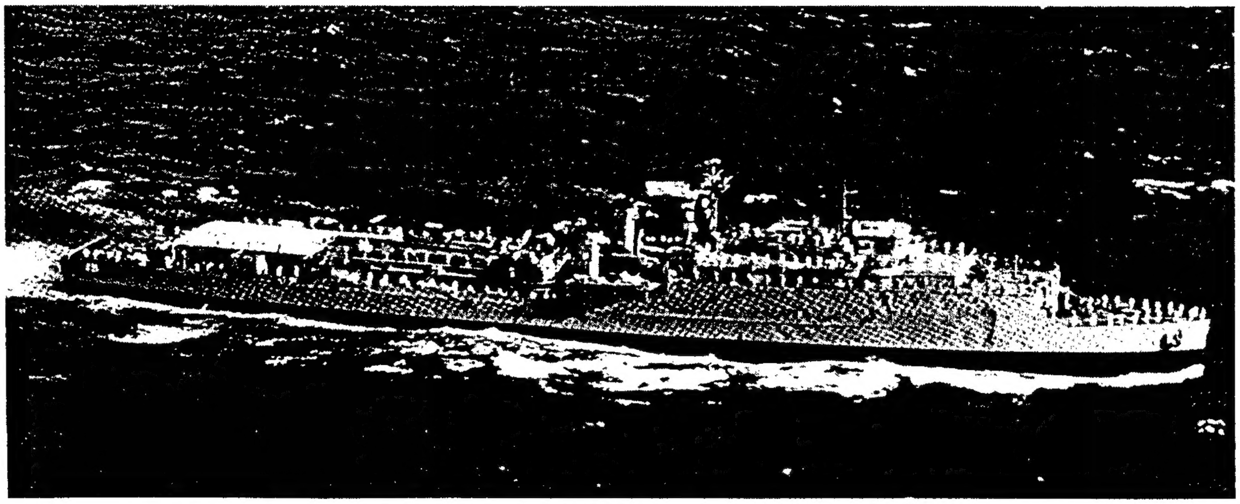
MICHAEL HAMMAN
DAVID C. WILKINS

Beckman Institute
University of Illinois, Urbana, Illinois

PATRICIA A. TATEM
FREDERICK W. WILLIAMS

Navy Technology Center for Safety and Survivability
Chemistry Division

December 19, 2001



REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE December 19, 2001	3. REPORT TYPE AND DATES COVERED Final FY 1999-FY 2001	
4. TITLE AND SUBTITLE Supervisory Control System for Ship Damage Control: Volume 7 — Software Architecture			5. FUNDING NUMBERS PE - 63508N
6. AUTHOR(S) Michael Hamman,* David C. Wilkins,* Patricia A. Tatem, and Frederick W. Williams			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory, Code 6180 4555 Overlook Avenue, SW Washington, DC 20375-5320			8. PERFORMING ORGANIZATION REPORT NUMBER NRL/MR/6180--01-8588
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 800 North Quincy Street Arlington, VA 22217-5660			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES *Beckman Institute, University of Illinois, Urbana, IL 61801			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE
13. ABSTRACT (*Maximum 200 words*) This report describes the system architecture of the Supervisory Control System for Ship Damage Control (DC-SCS). Its purpose is to provide for the reader an understanding of the structure of the software modules that make up DC-SCS and how these modules interrelate during the run-time of the system.			
14. SUBJECT TERMS Flooding, Fire, Damage control, Supervisory control, Automation			15. NUMBER OF PAGES 20
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

SUPERVISORY CONTROL SYSTEM FOR SHIP DAMAGE CONTROL: VOLUME 7 – SOFTWARE ARCHITECTURE

1. Overview

This report describes the system architecture of the Supervisory Control System for Ship Damage Control (DC-SCS). Its purpose is to provide for the reader an understanding of the structure of the software modules that make up DC-SCS and how these modules interrelate during the run-time of the system.

There are four primary modules of DC-SCS. These are:

1. Intelligent Reasoning Critiquing Learning **(IRCL)**
2. Graphic Visualization **(VIS)**
3. Human-Computer Interfaces **(HCI)**
4. Simulation and Scenario Generation **(SIM)**

These will each be described in detail.

2. Basic Terminology

2.1 System

The term "system" describes the entire software system that comprises the DC-SCS system.

2.2 Subsystem

A "subsystem" constitutes the highest functional level within the system that is not the system itself. The four subsystems that comprise DC-SCS are:

1. Intelligent Reasoning Critiquing Learning **(IRCL)**
2. Graphic Visualization **(VIS)**
3. Human-Computer Interaction **(HCI)**
4. Simulation and Scenario Generation **(SIM)**

2.3 Module

A "module" is a stand-alone executable that runs in its own process space. A module will either fall under the more general functionality of a subsystem, or it will perform some other support function, as for instance that which enables one subsystem to communicate with another.

2.4 Component

A "component" is a discrete part of a module. A module is composed of a number of components, each of which contributes some small functionality to the large functional whole of which a module is comprised.

Manuscript approved October 1, 2001.

2.5 Class

A "class" is the lowest level element in the architecture. It defines a single behavioral unit that provides a very particular set of services.

3. Architectural Overview

In order to orient the reader, it is perhaps best to begin with an overview of the entire system. The architectural overview will be presented in two stages: Level 1 presents the highest-level view; Level 2 presents a more detailed view.

In this overview, we consider the DCX system as run on-board the ex-USS *Shadwell* [Carhart, et.al., 1992].

3.1 Level 1

Figure 1 depicts this highest level view of the architecture.

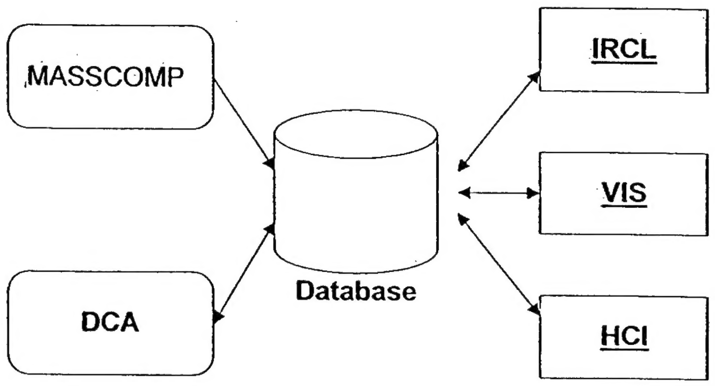


Figure 1. Architecture Level 1

The MassComp compiles data from sensors on the ship [Street et.al., 2000] and places these data into the database. Each subsystem retrieves data from the database in order to accomplish its particular set of tasks. All subsystems run concurrently, each in a separate process space. The database has a dual function: as a history repository (archiving events that have occurred) and to enable communication among ship, DC-SCS system, Damage Control Assistant (DCA) and other ship personnel.

The **IRCL** retrieves data from the database pertaining to the states of the ship's sensors. It probabilistically identifies casualty type and severity and then generates casualty response plans, which it places into the database. The **VIS** also retrieves ship sensor data from ship and updates, on a periodic basis, the appearance of the ship visualization based on data so retrieved. The **HCI** provides both input and output modalities. It retrieves casualty response data from the database (placed there by the **IRCL**) and displays them within the Action Window display for DCA observation and response. The **HCI** places responses from the DCA into the database, which the **IRCL** retrieves and incorporates into revised casualty response plans.

3.2 Level 2

Figure 2 presents a slightly lower level of architectural detail.

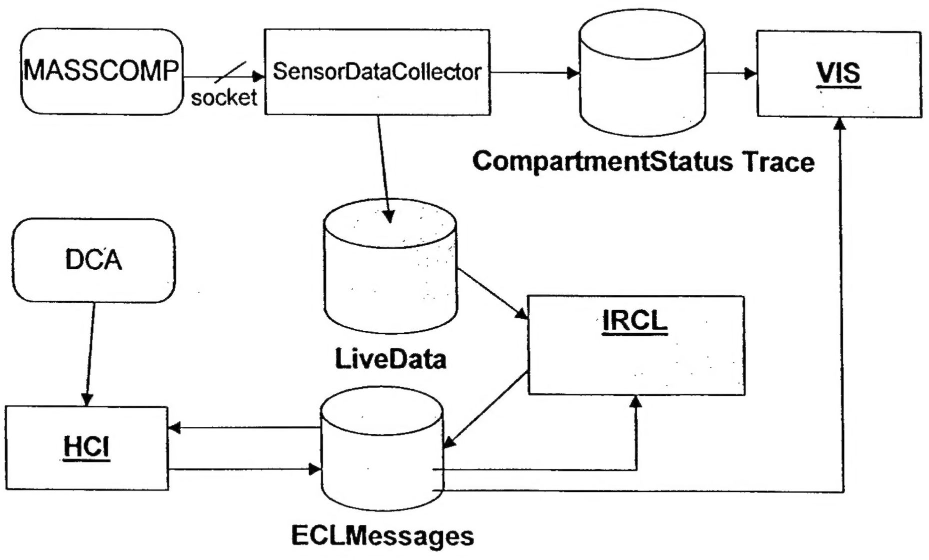


Figure 2. Architecture Level 2

The MassComp collects data from the ship's sensors and dispatches these data as discrete "events" over a socket. The **SensorDataCollector** is a distinct module that runs in its own process space. It connects to the MassComp socket, listening for and extracting event messages off that socket. It filters each event message, discarding events that most likely are the result of noise in the original sensor signal. Then, each filtered event message is placed into two database tables: *LiveData* and *CompartmentStatusTrace*. (For more information on the SensorDataCollector see Grois, E., et al., 2001.

The **VIS** subsystem polls data from the *LiveData* table and renders that data in the visualization. A Classification module within the **IRCL** subsystem polls and extracts ship data from the *LiveData* table. It probabilistically identifies casualty type from these data and places each such classification into the *ECLMessages* table. The CasualtyResponse module (also within the **IRCL** subsystem) retrieves this classification data from the *ECLMessages*. It assesses the severity of the event in terms of its impact on casualty response on the ship and estimates available resources (such as personnel and equipment), all in order to generate an appropriate casualty response plan of action. This plan is placed as a sequence of messages into the *ECLMessages* table.

The casualty response action messages are picked up from the *ECLMessages* table within the **HCI** subsystem and displayed in the "Action" Window. The 3-pane Action window allows the user (the DCA in the diagram) to read recommended actions from the **IRCL** and either execute those actions or instigate new actions. These actions are placed into the *ECLMessages* table, from which they are retrieved within the **IRCL** for further assessment and generation of new casualty response actions.

Meanwhile, the **VIS** periodically polls for and retrieves messages from the *ECLMessages* table in order to keep the ship visualization current.

4. Utilities

4.1 DCATimer

The DCATimer is a stand-alone module that places a **9102** message [Wilkins, D.C., et al., 2001] into the ECLMessages table every 10 seconds. This message is used by subsystems to keep track of the passage of time. For instance, the **SIM** subsystem uses it in order to know when to place simulated ship events into the database. The **VIS** uses it in order to update the state of the ship's visualization.

Figure 3 depicts the dataflow for DCATimer, and the subsystems, which retrieve their timer messages. The DCATimer starts an IDCXTimer object, which keeps track of the passage of time. Every 10 seconds, the DCATimer obtains the current time from the IDCXTimer and posts that time as a **9102** message to the *ECLMessages* table.

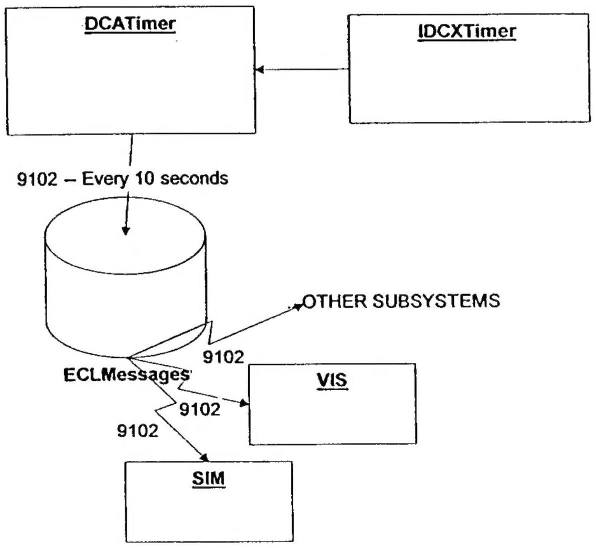


Figure 3. DCATimer data flow

5. IRCL Subsystem

Currently, the **IRCL** subsystem consists of two modules. One module—the Classifier—analyzes ship data and classifies events on the ship (such as fire and flood) using probabilistic techniques. The other module—CasualtyResponse—assesses the severity of each such event and intelligently formulates an action plan in response to that event, which it makes available—through the **HCI**—to the DCA.

Figure 4 depicts the data flow, from the MassComp to the **IRCL** modules.

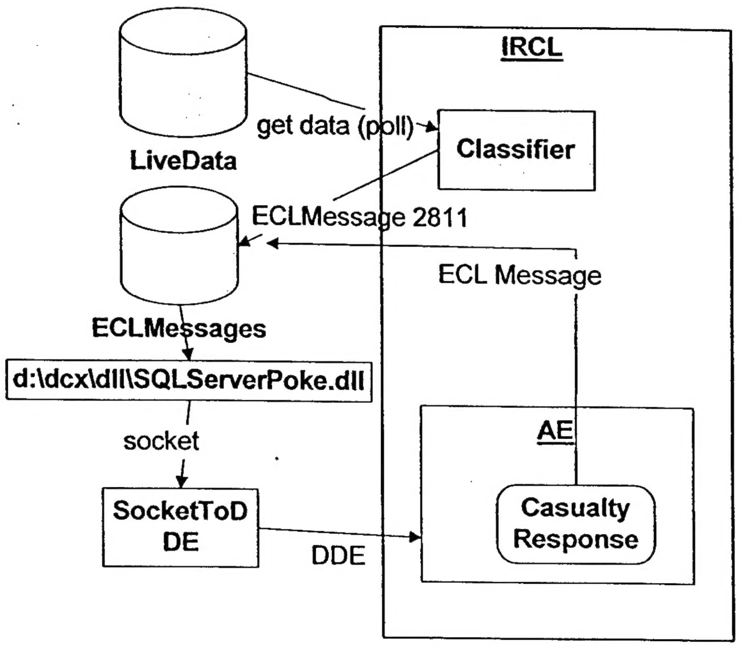


Figure 4. IRCL data flow

The **Classifier** polls the LiveData table for events, probabilistically determining a classification to describe an event such as "fire," "flood," and so on. Each such classification is formulated as an "ECL" message and placed into the *ECLMessages* table. "ECL" stands for event communication language. The ECL constitutes a lexicon of "events" that will be of interest to different modules in the system. ECL messages are placed into the *ECLMessages* table. Any module that cares about the current state of the ship (or other aspects of the system, including states within particular modules) monitors the *ECLMessages* table, extracting those ECL messages that are of particular interest to it.

In some cases, a given module will poll the database, while in other cases data will be pushed from the database into the module. Using a push method, triggers are attached to a database table. A trigger "fires" whenever a new row is added to the table. This trigger launches an SQL script that invokes a function within a DLL (dynamically linked library) specified in the SQL script. Here the DLL function takes the data just inserted into the database and creates a text-based message which it sends out on a socket. The **SocketToDDE** module periodically scans that socket and formats incoming socket messages as DDE messages to be dispatched to Art Enterprise. Microsoft Windows Dynamic Data Exchange (DDE) is an inter-process communication mechanism supported by the Windows API.

The **Casualty Response** module runs inside Art Enterprise (AE). Art Enterprise includes a mechanism that allows other software components to send messages to it using DDE. Art Enterprise forwards these messages to any component running in its environment that is interested in them, which in the current case would be the **Casualty Response** module. As already described (3.2), the **Casualty Response** assesses the severity of the event in terms of its impact on casualty response on the ship and estimates available resources (such as personnel and equipment). The resulting action plan is placed as a sequence of messages into the *ECLMessages* table.

6. HCI Subsystem

The **HCI** subsystem consists of two lower level sets of modules:

1. DCInterface
2. Components

Each will be discussed in some detail below.

The overall architecture for the **HCI** subsystem is depicted in Figure 5:

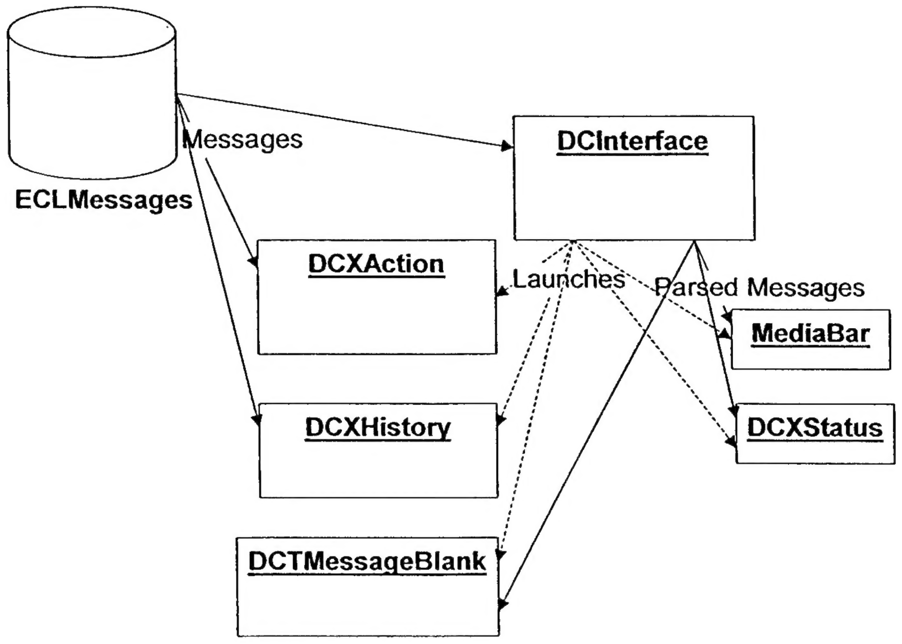


Figure 5. HCI architecture—High-level view

As is shown DCInterface launches DCXAction, DCXHistory, DCXStatus, and DCTMessageBlank. This is depicted by the dashed lines in Figure 5.

DCXAction, DCXHistory, DCTMessageBlank and DCXStatus are all Component Object Model (COM) objects. DCXAction displays the DCA 3-pane action window. The DCXHistory displays messages in the "history" window. The DCTMessageBlank object displays the Command Menu window. DCXStatus displays each message in a status window.

Messages are directly obtained from the *ECLMessages* table by DCXAction and DCXHistory modules. Meanwhile, DCInterface polls ECLMessages and parses messages within a particular range and passes the parsed messages to MediaBar, DCXStatus, and DCTMessageBlank objects.

Architecturally, the **HCI** subsystem can be broken down into two lower level sets of modules:

1. DCInterface
2. Components (DCXAction, DCXHistory, DCTMessageBlank)

Each will be discussed in some detail below.

6.1 DC Interface

DC Interface is the windowing structure that launches and contains all of the **HCI** windows components. In addition, it parses messages which it extracts from the ECLMessages table and, if these are of the appropriate type, passes parsed messages to the following modules:

- DCXStatus
- MediaBar

6.2 Components

6.2.1 DCXAction

DCXAction creates and handles interactions with the 3-pane Action window. The three panes are "Pending Actions" and "Overridden Actions." Figure 6 depicts the data flow into and out of DCXAction. **CasualtyResponse** formulates a proposed action which is in response to the occurrence of a crisis event on the ship and laces this proposed action as a message into the ECLMessages table. These messages will be in the range 1100 – 1200 (Wilkins, D.C., et al., 2001). DCXAction polls the ECLMessages table for messages in this range, displaying any such messages in the "Pending Action" window pane. Meanwhile, if the DCA drags a message from the "Pending Action" pane into the "Active Actions" pane, that message is dispatched in the ECL Messages table. From here it is picked up by the DCXHistory object, which displays the message in the History window.

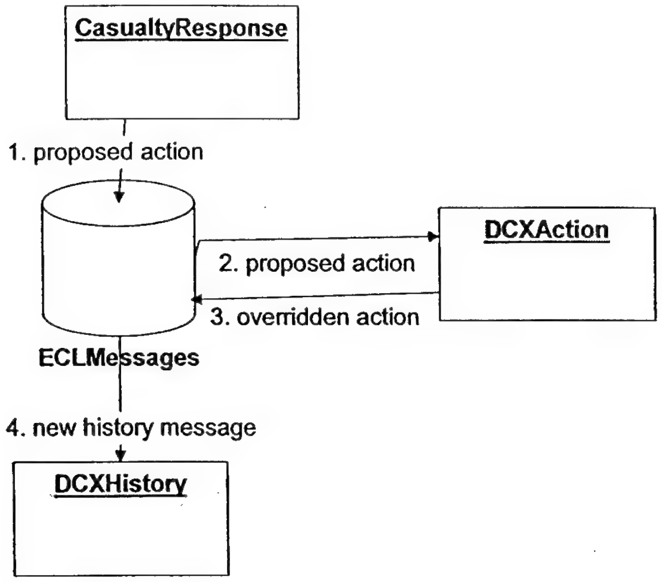


Figure 6. DCXAction data flow

6.2.2 DCX History

The DCXHistory module causes all of the ECL messages to be displayed within a pane inside the DCInterface window. The data flow is depicted in Figure 7.

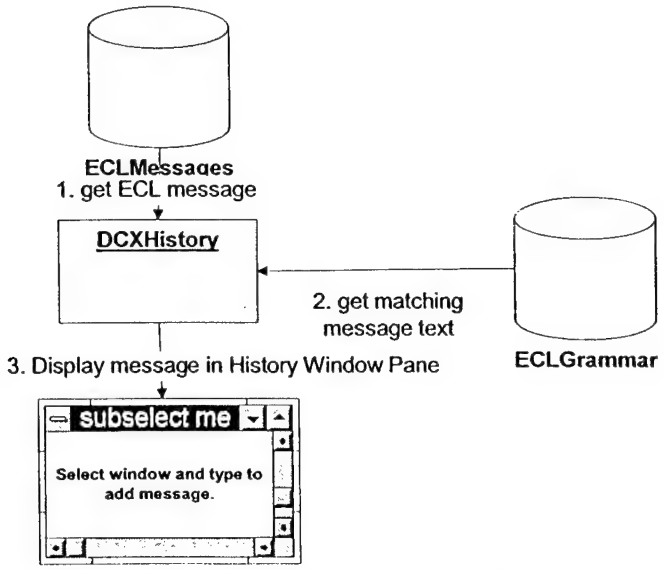

ECLMessages
1. get ECL message
DCXHistory
2. get matching message text
ECLGrammar
3. Display message in History Window Pane
subselect me
Select window and type to add message.

Figure 7. DCXHistory data flow

DCXHistory polls the *ECLMessages* table for new messages. When it finds new messages there, it locates the matching ECL message grammar string by pulling it from the *ECLGrammar* table. It then displays that message in the History Window Pane of the DCInterface.

7. VIS Subsystem

7.1 Initialization

Figure 8 depicts a high-level data-flow view of the initialization of the **VIS** subsystem. The **VIS** loads data from four database tables: *Vertices, Sides, Walls, and Compartments*. After loading these data, it creates a visualization of the ship using these data.

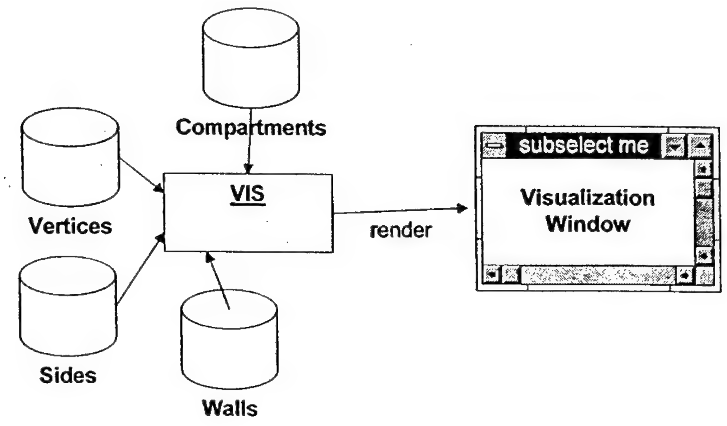


Figure 8. High-level data flow for initialization of VIS

7.2 Run-Time

The **VIS** subsystem is composed of four distinct modules. These are:

1. ES—Event Messaging System
2. DB—The database communication layer that handles getting messages from the database
3. FRAMEWORK—the **VIS** "framework" which acts as a kind of director, directing events from the outside world (the database) to the visualization
4. AG—visualization rendering (using OpenGL)

These modules will be discussed in some detail.

7.2.1 Event Message

The Event Message System defines the Event messages that are dispatched **by the DB** to the **Framework**. There are two primary classes involved in implementing the Event Message system. They are depicted in Figure 9.

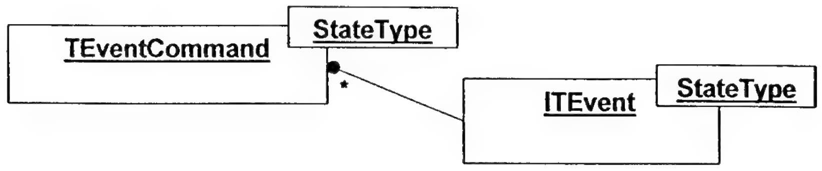


Figure 9. Event message

ITEvent is the event primitive class whose template argument is the type of the state of the event (e.g. fire, flood, etc.). TEventCommand contains a vector of ITEvent objects.

7.2.2 Database Polling Module

The **DB** module polls tables within the database for relevant data. When new data are presented, the **DB** module retrieves those data (a particular row in a table) and packages them as an "event," which it then dispatches to the **Framework** module. The **DB** and **Framework** modules run in separate threads. This is depicted in Figure 10.

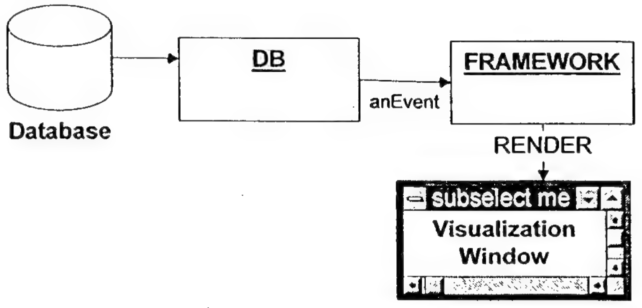


Figure 10. Database polling for the VIS

At a lower level of detail, the **DB** and **Framework** modules run in separate threads. The DB module periodically polls the *ECLMessages* and *CompartmentStatusTrace* tables. For each access to the database, newly inserted rows are retrieved. For each such row, an ITEvent object is created and added to a TEventCommand object (see 7.3 for a descriptions of these classes). The resulting TEventCommand object is dispatched to the **Framework**, by invoking the appropriate member function call. The rendering engine (AG) takes these data and updates the ship visualization.

The data flow is depicted in Figure 11.

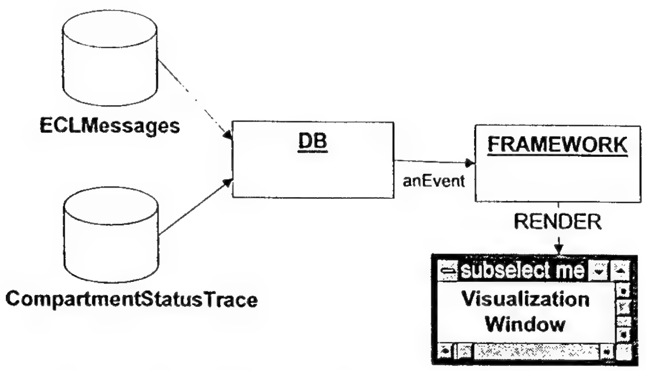


Figure 11. DB and Framework threads

7.2.3 Framework

The **Framework** module defines the mechanism though which data are retrieved from the database and is dispatched to the display to act as the primary system entry point to the **VIS.** The **Framework** contains two primary sets of components:

1. The DCTVisCtl component
2. The User Interface components

The DCTVisCtl component maintains a handle to an object that handles graphical visualization of the ship (which is defined in the AG module). The DCTVisCtl delegates Windows events, as well as events dispatched to the **Framework** from the **DB**, to the ShipView and ShipWindow objects. This activity is depicted in Figure 12.

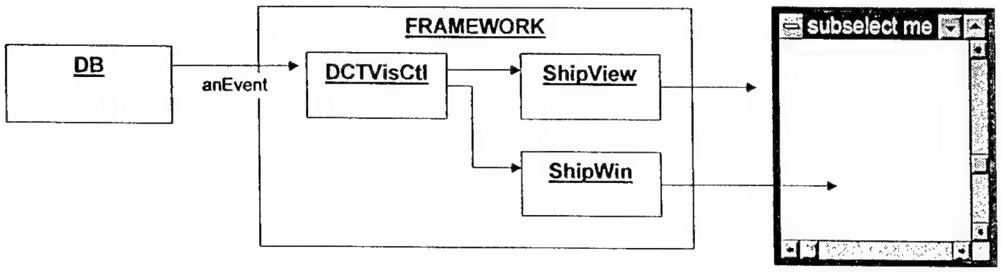


Figure 12. DCTVisCtl

The ShipView handles messages to the window itself. These include resizing, repositioning, and other such messages to the window. The ShipWin has a handle to the Graphics Engine, which contains all of OpenGL code for rendering the visualization within the window.

7.2.4 Graphics Engine (AG)

The Graphics Engine (AG) renders the ship visualization. Figure 13 shows the static class structure for classes involved in rendering.

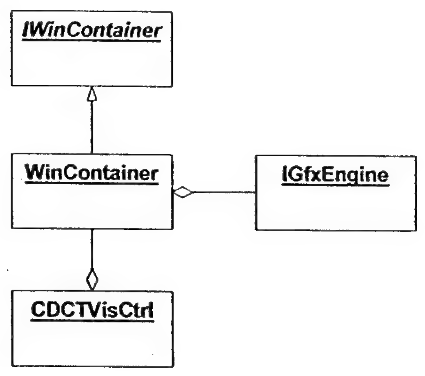


Figure 13. AG class structure

IWinContainer is an abstract class ("interface"). WinContainer implements the IWinContainer interface—it "contains" the graphical rendering view inside the main visualization window, which is defined by the CDCTVisCtl class. As such, the CDCTVisCtl contains a reference to the WinContainer object. The WinContainer in turn has a pointer to the Graphics Engine object (IGfxEngine class) to which it delegates MSWindows paint messages. The IGfxEngine class encapsulates all of the OpenGL library calls.

The dynamic behavior of a paint message is depicted in Figure 14.

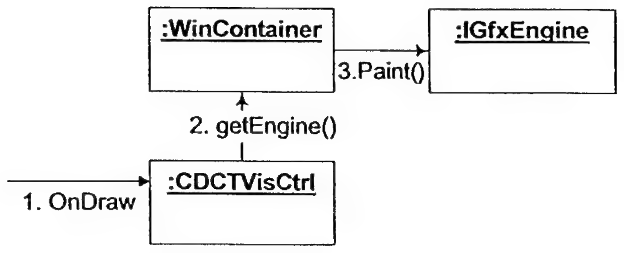


Figure 14. Painting the Visualization window

Each time the Visualization window needs to be redrawn, the *OnDraw()* method of the CDCTVisCtl object is invoked. Here the call is delegated to the WinContainer object, which in turn tells the IGfxEngine to paint itself. The IGfxEngine then re-renders the visualization using OpenGL library routines that are encapsulated in the IGfxEngine class.

8. SIM Subsystem

The Simulation subsystem consists of two rather large modules: The **Simulation (SIM)** module and the **Scenario Generation (ScenGen)** module. The **ScenGen** (1) assists the user in defining a particular scenario (though a GUI interface) and (2) generates a timed sequence of events on the basis which the **SIM** module computes and generates Ship-related events. Figure 15 depicts a high level view of this structure.

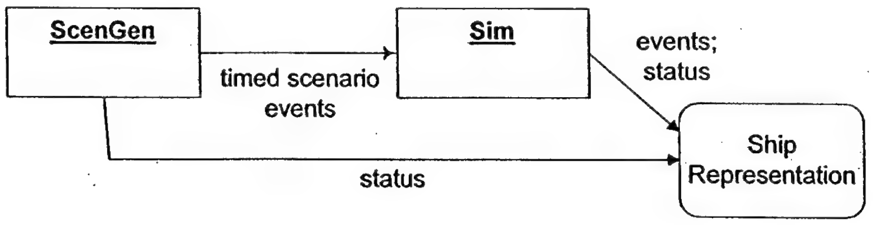


Figure 15: High-level data flow for scenario generation and simulation

Figure 16 depicts a lower level of data flow:

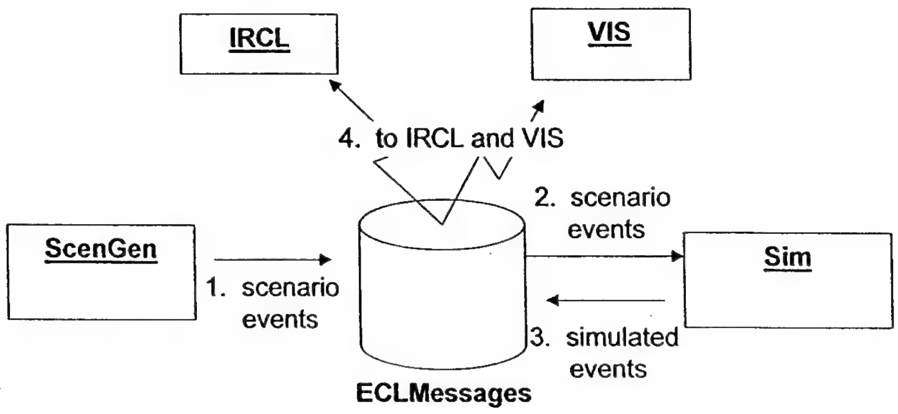


Figure 16: Data flow for scenario generation and simulation

First, **ScenGen** generates a sequence of scenario events and places these into the *ECLMessages* table. Each such event has a scheduled time attached to it. The **SIM** scans these messages and, when the scheduled time for a given event matches the current time in the **SIM**, the **SIM** retrieves that event from the *ECLMessages* table. The **SIM** generates its own set of events based upon this event. These newer events constitute a sequence of physical states attributed to the ship. These events are placed into the *ECLMessages* table from which they are retrieved by the **IRCL** and the **VIS**.

In addition, both the **ScenGen** and the **SIM** place data into other tables (e.g. the Doors table). These data are also retrieved by the **IRCL** and the **VIS**.

8.1 Scenario Generation (ScenGen)

Figure 17 displays a most general view of the ScenGen class structure—many details are left out in order to convey the architecturally significant aspects.

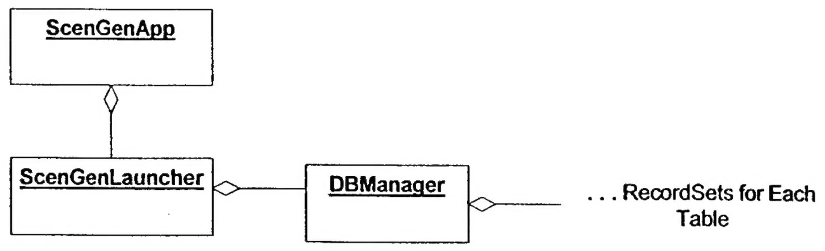


Figure 17. Scenario generation

The ScenGenApp is the main application class for the ScenGen. It contains a ScenGenLauncher object. The ScenGenLauncher has the responsibility of loading a particular scenario (as defined by the user through the user interface of the scenario generator) and of running the scenario loaded.

The ScenGenLauncher contains, in turn, a DBManager object. The DBManager manages all database access functionality. It contains record sets for each table which the ScenGen accesses.

8.1.1 Running the Scenario

Figure 18 is a behavioral model of the **ScenGen** during startup and runtime. First the user enters in database and scenario selections. Then the ScenGenApp creates a new ScenGenLauncher, initializing it with the database information entered. The Scenario is then loaded ("LoadScenario()") and run ("RunScenario()").

When the scenario is first run, *StartSimulation()* is invoked against the DBManager object. This results in a 9110 message being placed into the *ECLMessages* table. The **SIM**, which is polling the ECLMessages table, retrieves this message and starts running the simulation when it does so.

Meanwhile, the ScenGenLauncher continues running. It pulls each action from the scenario list of actions, in time sequential order. For each sequential action retrieved, ScenGenLauncher sleeps until the time scheduled for that action occurs. Then the body of the action is executed. Executed actions result in messages being placed into the database, which will be picked up by the **SIM**.

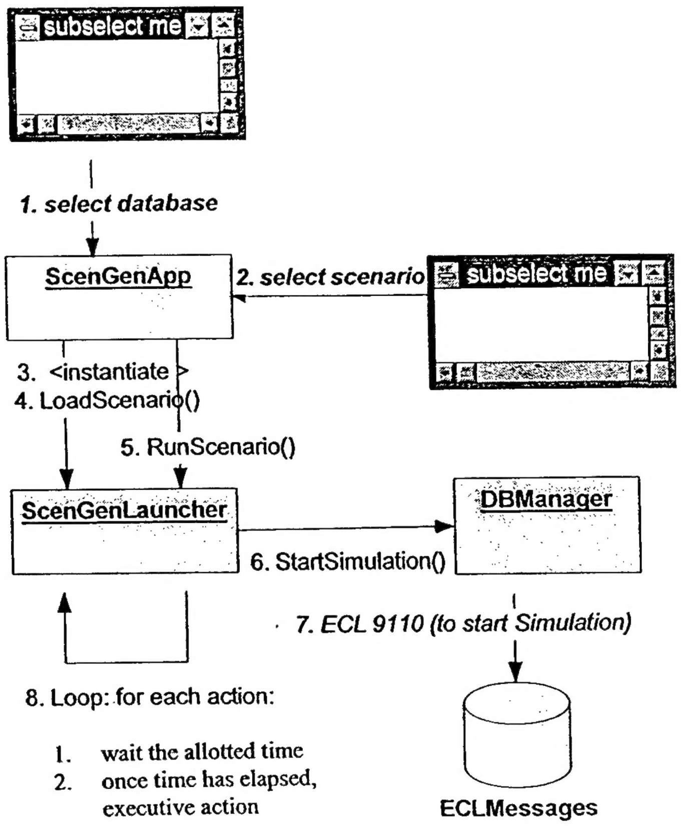


Figure 18. ScenGen startup and runtime behavioral model

8.2 Simulation

Figure 19 shows the static class model for part of the SIM. In the interest of architectural clarity, details are left out as are the user interface classes.

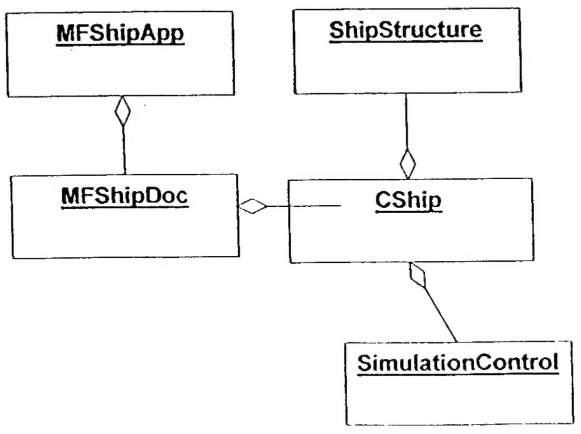


Figure 19. Static class model of SIM

MFShipApp is the main application class for the **SIM**. MFShipDoc is the document class, which handles the data aspect of the MFShipApp. Cship is the ship representation. It has a SimulationControl and a ShipStructure object. The SimulationControl handles all simulation control activities, while the ShipStructure maintains a representation of the structure of the ship.

8.2.1 Running the Simulation

Figure 20 shows the run-time for the simulation.

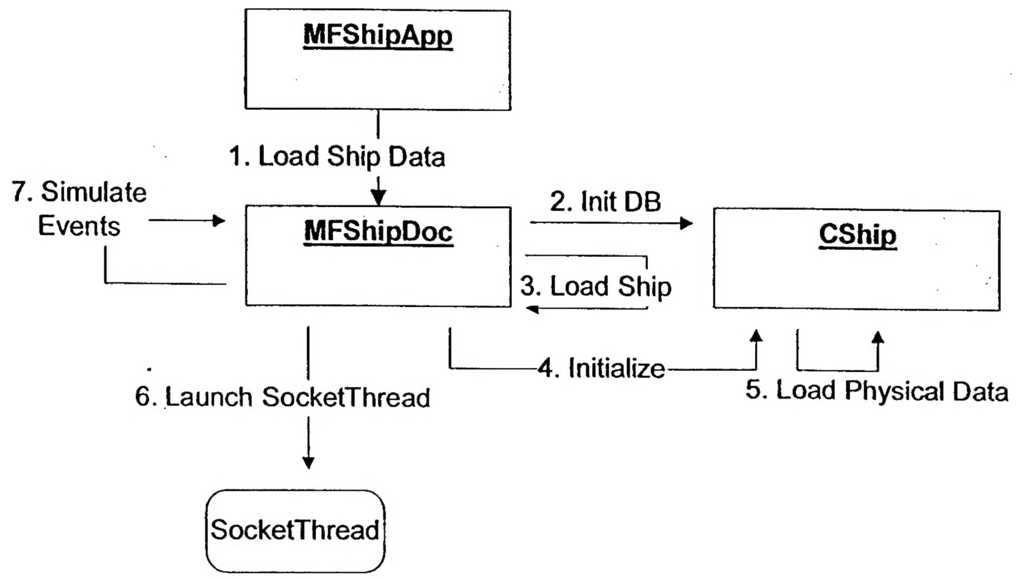


Figure 20. Run-time simulation

As shown, the MFShipApp begins by telling the MFShipDoc to load data. This might be done through the user interface or through initialization at the command line. The MFShip tells the Cship to initialize the database; it then does the actual loading of ship data, which it passes to the Cship object in step #4. The Cship loads all of the physical ship data. Once the database has been initialized and the ship data loaded, the MFShipDoc launches a thread, which will dispatch events from the simulated ship over a public socket. While this thread runs, the MFShipDoc runs the simulation that produces the events that are dispatched over the socket.

During simulation, the **VIS** subsystem connects to this socket and continuously reads events from it to update its ship rendition.

9. References

1. Carhart, H.W., Toomey, T.A. and Williams, F.W., "The Ex-USS SHADWELL Full-Scale Fire Research and Test Ship," NRL Memorandum Report 6074 of 6 October 1982, reissued September 1992

2. Grois, E., Wilkins, P.C., Bearman, I., Bobak, M., Brady, A., Hebble, P., Miller, M., Sitter, S., Tatem, P.A. and Williams, F.W., "Supervisory Control System for Ship Damage Control: Volume 4 – Intelligent Reasoning," NRL Memorandum Report NRL/MR/6180—01—8583, September 28, 2001

3. Street, T.T., Bailey, J., Riddle, T., Tate, D. and Williams, F.W., "Up-grades to Data Handling Capabilities on ex-USS SHADWELL," NRL Ltr Rpt 6180/0229 of 6 June 2000

4. Wilkins, D.C., Shultz, K., Daniels, M., Carbonari, R., Shou, G., Spillner, B., Gimbel, K., Bulitko, V. Tatem, P.A. and Williams, F.W.," Supervisory Control System for Ship Damage Control: Volume 5 – Knowledge Ontology, " NRL Memorandum Report, NRL/MR/6180—01—8573, August 24, 2001